

Introduction to (Unconventional) Vectorization

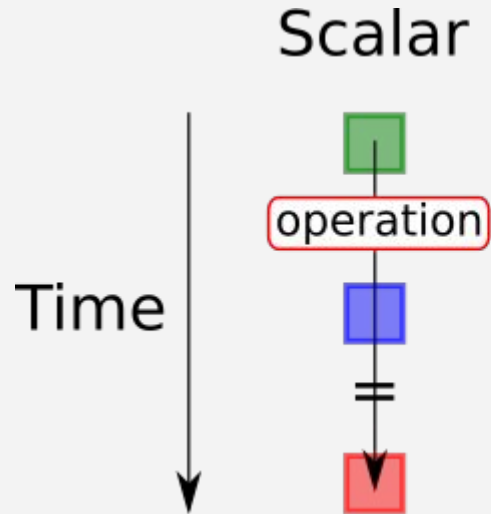
Stefanos Baziotis

NEC Deutschland

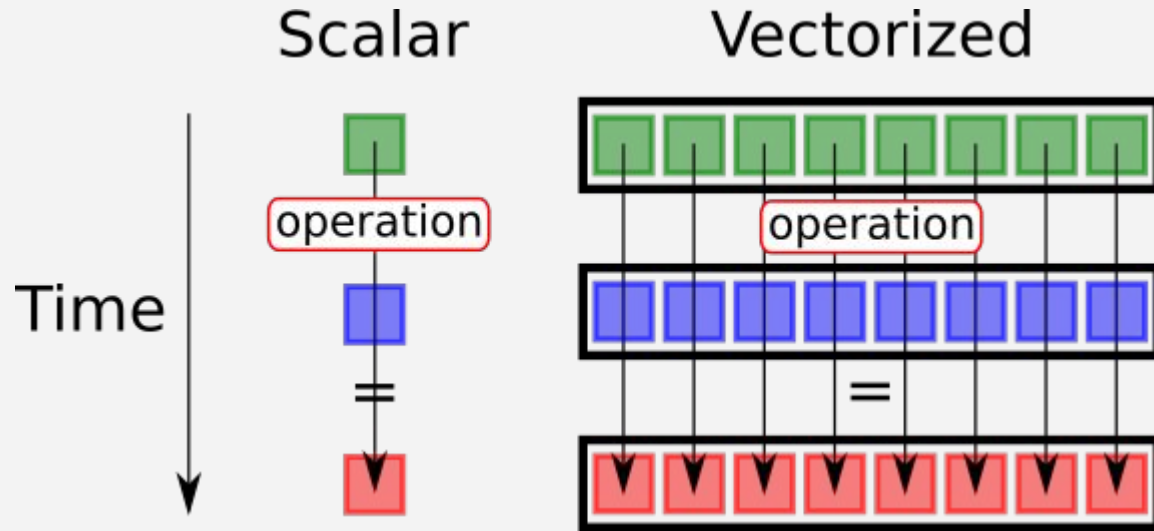
National and Kapodistrian University of Athens

stefanos.baziotis@gmail.com

What is Vectorization ?



What is Vectorization ?



Vectorized Code in a High-Level Language

```
int add(int a, int b) {  
    return a + b;  
}
```

Intrinsics

```
__m128i add4(__m128i a, __m128i b) {  
    return _mm_add_epi32(a, b);  
}
```

Vectorizing a Loop

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; ++i)  
        C[i] = A[i] + B[i];  
}
```

Vectorizing a Loop

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; i += 4) {  
        v1 = load 4 values from A  
        v2 = load 4 values from B  
        res = 4-packed add(v1, v2)  
        store res (which is 4 values) in C  
    }  
}
```

Loading a Vector

```
_mm_loadu_si128((const __m128i*) m);
```


Loading a Vector

```
__m128i loadu_si128(void *m) {  
    return _mm_loadu_si128((const __m128i*) m);  
}
```

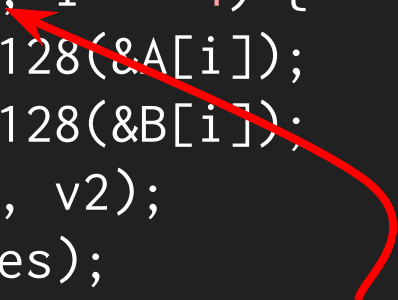
Vectorizing a Loop

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; i += 4) {  
        __m128i v1 = loadu_si128(&A[i]);  
        __m128i v2 = loadu_si128(&B[i]);  
        __m128i res = add4(v1, v2);  
        storeu_si128(&C[i], res);  
    }  
}
```

The Struggles of Vectorization

Vectorizing a Loop

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; i += 4) {  
        __m128i v1 = loadu_si128(&A[i]);  
        __m128i v2 = loadu_si128(&B[i]);  
        __m128i res = add4(v1, v2);  
        storeu_si128(&C[i], res);  
    }  
}
```



What if len is not a multiple of 4?

Vectorizing a Loop

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    int end = (len % 4 == 0) ? len : len - 4;  
    for (int i = 0; i < end; i += 4) {  
        __m128i v1 = loadu_si128(&A[i]);  
        __m128i v2 = loadu_si128(&B[i]);  
        __m128i res = add4(v1, v2);  
        storeu_si128(&C[i], res);  
    }  
}
```

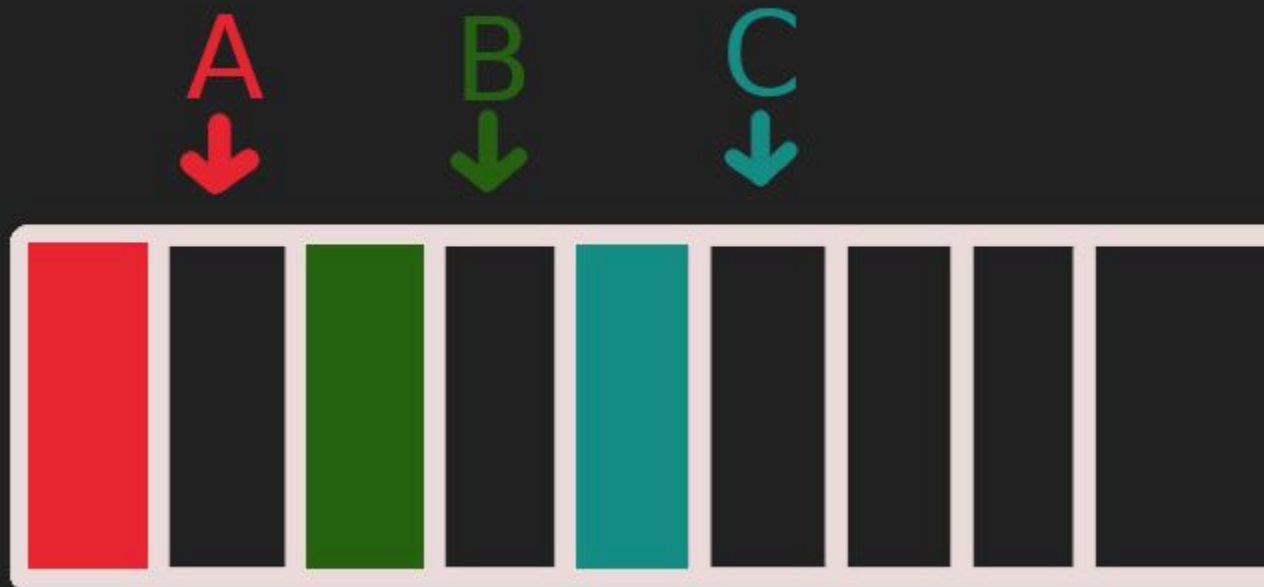
Vectorizing a Loop

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    int end = (len % 4 == 0) ? len : len - 4;  
    int i = 0;  
    for (; i < end; i += 4) {  
        __m128i v1 = loadu_si128(&A[i]);  
        __m128i v2 = loadu_si128(&B[i]);  
        __m128i res = add4(v1, v2);  
        storeu_si128(&C[i], res);  
    }  
    for (; i < len; ++i) {  
        C[i] = A[i] + B[i];  
    }  
}
```

Aliasing



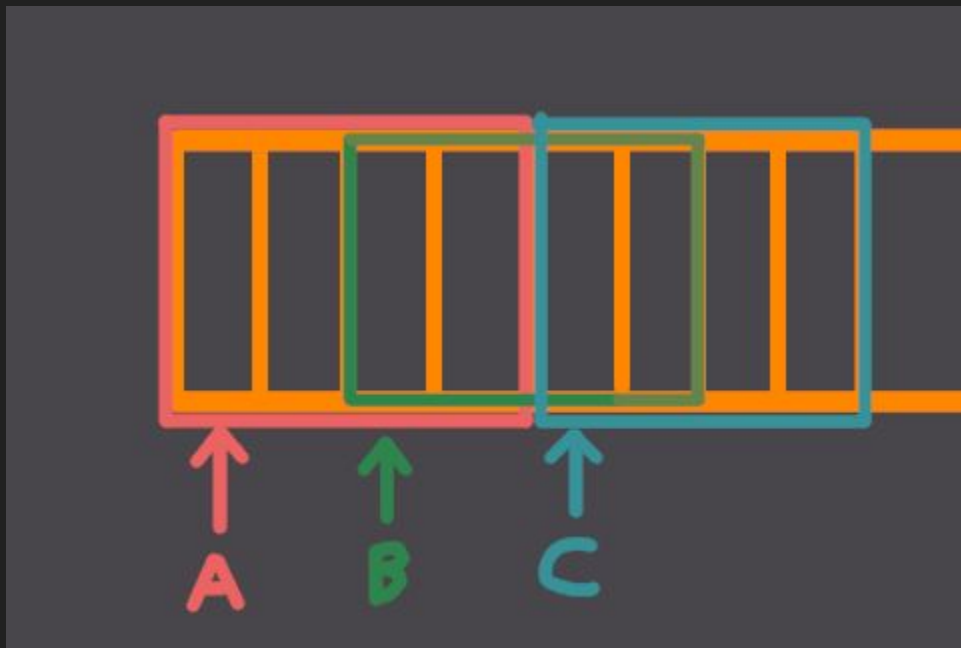
Aliasing



Aliasing



Aliasing



Aliasing

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    if (A + len > B)  
        jump to the scalar version  
    if (B + len > A)  
        jump to the scalar version  
    ...  
}
```

Aliasing

```
void sum_arrays(int *restrict A,  
               int *restrict B,  
               int *restrict C,  
               int len) {  
    ...  
}
```

Reductions

Reductions

```
new_value = operation(old_value, data);
```

Accumulation

```
int sum_of_array(int *a, int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i)  
        sum = sum + a[i];  
}
```

Accumulation

```
int sum_of_array(int *a, int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i)  
        sum = sum + a[i];  
}
```

new value



Accumulation

```
int sum_of_array(int *a, int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i)  
        sum = sum + a[i];  
}
```

new value

old value

Accumulation

```
int sum_of_array(int *a, int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i)  
        sum = sum + a[i];  
}
```

new value

old value

operation

Accumulation

```
int sum_of_array(int *a, int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i)  
        sum = sum + a[i];  
}
```

new value

old value

operation

data


Associativity and Commutativity

```
new_value = operation(old_value, data);
```

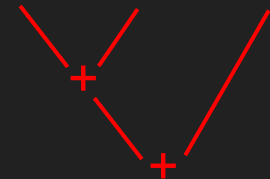
Associativity

$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$

Associativity

$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$


Associativity

$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$


Associativity

$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$

The diagram illustrates the evaluation order of the expression $((1 + 2) + 3) + 4$. Red lines connect the numbers to the addition operators in a tree structure, showing the sequence of operations from left to right. The first addition is $1 + 2$, followed by $(1 + 2) + 3$, and finally $((1 + 2) + 3) + 4$. The right-hand side of the equation, $(1 + 2) + (3 + 4)$, represents the same result using the associative property of addition.

Associativity

$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$

The diagram illustrates the associativity of addition. On the left side of the equation, the expression $((1 + 2) + 3) + 4$ is shown. The sub-expressions $1 + 2$ and 4 are highlighted with red boxes. Red lines connect the $+$ signs to show the order of operations: first $1 + 2$, then $(1 + 2) + 3$, and finally $((1 + 2) + 3) + 4$. On the right side, the expression $(1 + 2) + (3 + 4)$ is shown. The sub-expressions $1 + 2$ and $3 + 4$ are highlighted with red boxes. Red lines connect the $+$ signs to show the order of operations: first $1 + 2$ and $3 + 4$, then $(1 + 2) + (3 + 4)$.

Associativity

$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$

The diagram illustrates the associativity of addition by showing two equivalent ways to compute the sum of four numbers: 1, 2, 3, and 4. The left side of the equation, $((1 + 2) + 3) + 4$, is represented by a tree structure where 1 and 2 are added first, then 3 is added to that result, and finally 4 is added to the total. The right side, $(1 + 2) + (3 + 4)$, shows 1 and 2 added together, and 3 and 4 added together, with the two intermediate results then being added to each other. Red boxes highlight the sub-expressions $(1 + 2)$, 3 , 4 on the left, and 1 , 2 , $(3 + 4)$ on the right. Red lines connect the numbers to the addition operators, showing the flow of the calculation.

Associativity

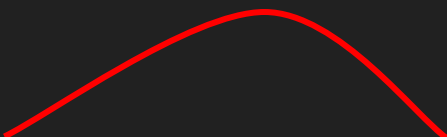
$$(((1 + 2) + 3) + 4) = (1 + 2) + (3 + 4)$$

The diagram illustrates the associativity of addition by comparing two different ways to compute the sum of four numbers: 1, 2, 3, and 4. The left side of the equation, $((1 + 2) + 3) + 4$, is represented by a binary tree where the root node is a '+' operator. Its left child is another '+' operator, which has children 1 and 2. The root's right child is a '+' operator with children 3 and 4. The right side of the equation, $(1 + 2) + (3 + 4)$, is represented by a binary tree where the root node is a '+' operator. Its left child is a '+' operator with children 1 and 2, and its right child is a '+' operator with children 3 and 4. In both trees, the numbers 1, 2, 3, and 4 are highlighted in red boxes, and the '+' operators are shown in red.

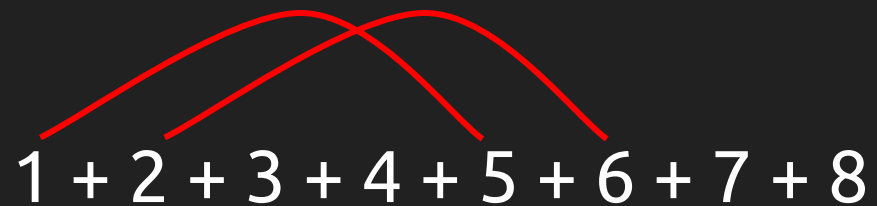
Commutativity

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

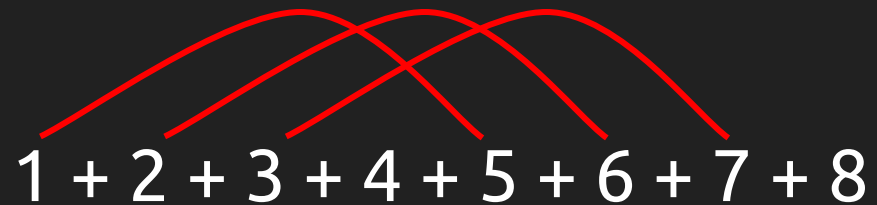
Commutativity


$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

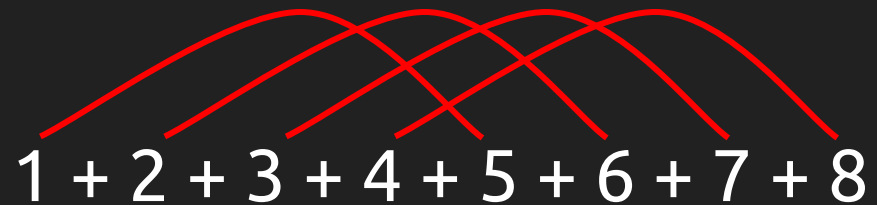
Commutativity


$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

Associativity


$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

Commutativity


$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

Loading from Memory

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

Loading from Memory

1 | 2 | 3 | 4

+

5 | 6 | 7 | 8

Vectorizing Summation

```
int sum_of_array(int *A, int len) {
    int end = (len % 4 == 0) ? len : len - 4;
    int i = ;
    __m128i acc = // Register with all lanes equal to 0;
    for (int i = 0; i < end; i += 4) {
        __m128i v = loadu_si128(&A[i]);
        acc = add4(v);
    }
    int final_result = add_all_lanes_epi32(acc);
    for (; i < len; ++i) {
        final_result += a[i];
    }
    return final_result;
}
```

Horizontal Add

```
int get_low_32_bits_si128(__m128i v) {  
    return _mm_cvtsi128_si32(v);  
}
```

```
int add_all_lanes_epi32(__m128i v) {  
    v = _mm_hadd_epi32(v, v);  
    v = _mm_hadd_epi32(v, v);  
    return get_low_32_bits_si128(v);  
}
```

Vectorizing Conditional Code

Invariant Condition

```
int a[VECTOR_WIDTH];  
int b[VECTOR_WIDTH];  
int out[VECTOR_WIDTH];  
...  
if (c) {  
    for (int i = 0; i < VECTOR_WIDTH; ++i)  
        out[i] = a[i] + b[i];  
}
```

Invariant Condition

```
__m128i a_vector = _mm_loadu_si128(&a[0]);  
__m128i b_vector = _mm_loadu_si128(&b[0]);  
if (c) {  
    __m128i res = add4(a_vector, b_vector)  
    _mm_storeu_si128(&out[0], res);  
}
```

Divergent Condition

```
int a[VECTOR_WIDTH];
int b[VECTOR_WIDTH];
int out[VECTOR_WIDTH];
...
for (int i = 0; i < VECTOR_WIDTH; ++i) {
    if (c[i]) {
        out[i] = a[i] + b[i];
    }
}
```


Divergent Condition

```
int a[VECTOR_WIDTH];
int b[VECTOR_WIDTH];
int out[VECTOR_WIDTH];
...
for (int i = 0; i < VECTOR_WIDTH; ++i) {
    if (c[i]) {
        out[i] = a[i] + b[i];
    }
}
```

Divergent!



Generic Form

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    if (cond[i]) { // Divergent condition  
        value[i] = some computation;  
    } else {  
        value[i] = some other computation;  
    }  
}
```

Predication

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    a[i] = some computation;  
}
```

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    b[i] = some other computation;  
}
```

Predication

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    a[i] = some computation;  
}
```

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    b[i] = some other computation;  
}
```

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    if (cond[i]) { // Divergent condition  
        value[i] = a[i];  
    } else {  
        value[i] = b[i];  
    }  
}
```

Predication

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    a[i] = some computation;  
}
```

1 instruction

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    b[i] = some other computation;  
}
```

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    if (cond[i]) { // Divergent condition  
        value[i] = a[i];  
    } else {  
        value[i] = b[i];  
    }  
}
```

Predication

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    a[i] = some computation;  
}
```

1 instruction

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    b[i] = some other computation;  
}
```

1 instruction

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    if (cond[i]) { // Divergent condition  
        value[i] = a[i];  
    } else {  
        value[i] = b[i];  
    }  
}
```

Predication

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    a[i] = some computation;  
}
```

1 instruction

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    b[i] = some other computation;  
}
```

1 instruction

```
for (int i = 0; i < VECTOR_WIDTH; ++i) {  
    if (cond[i]) { // Divergent condition  
        value[i] = a[i];  
    } else {  
        value[i] = b[i];  
    }  
}
```

1 instruction!!

Outer-Loop Vectorization

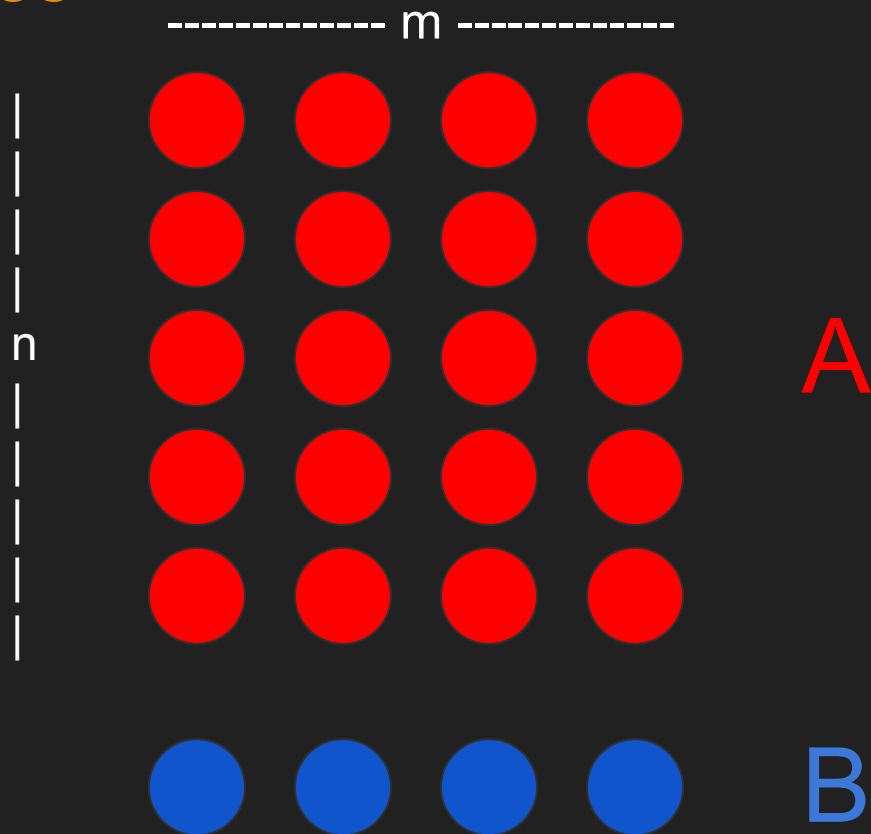
Simon Moll et al.

Nuzman, Zaks

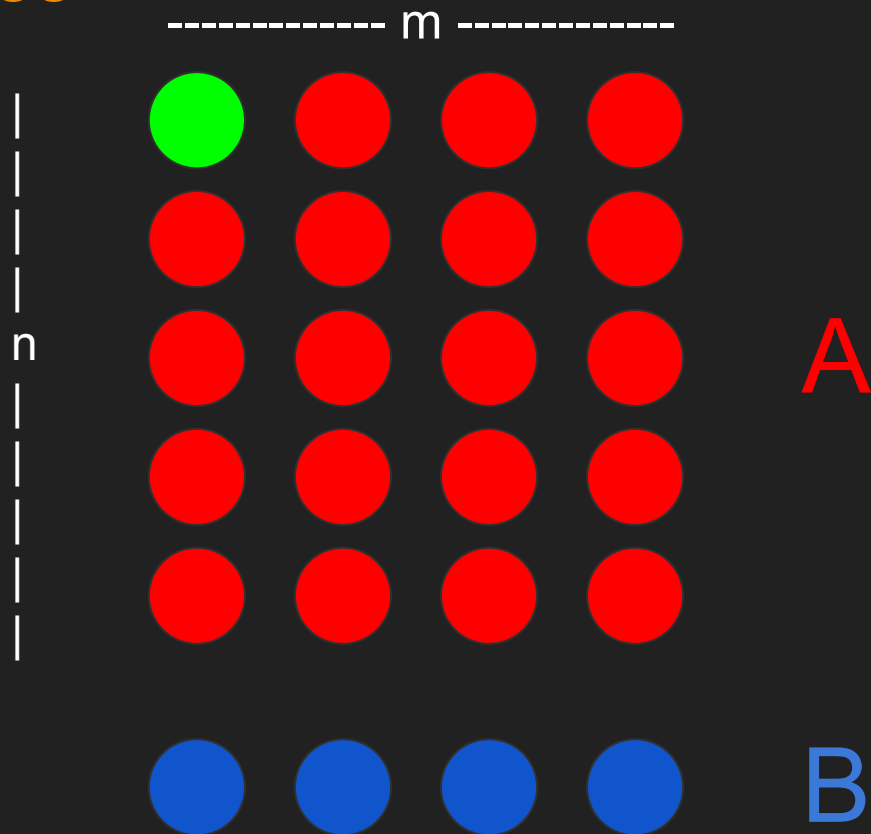
Outer-Loop Vectorization

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        int v = A[j*m + i];  
        a += v;  
    }  
    B[i] = a;  
}
```

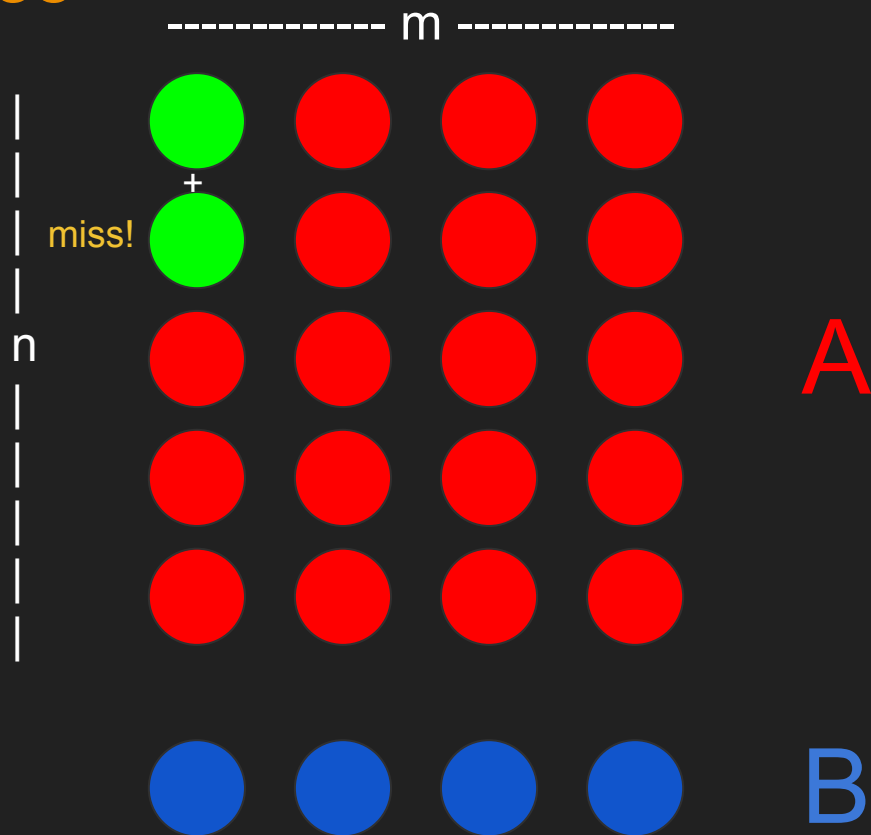
Iteration Space



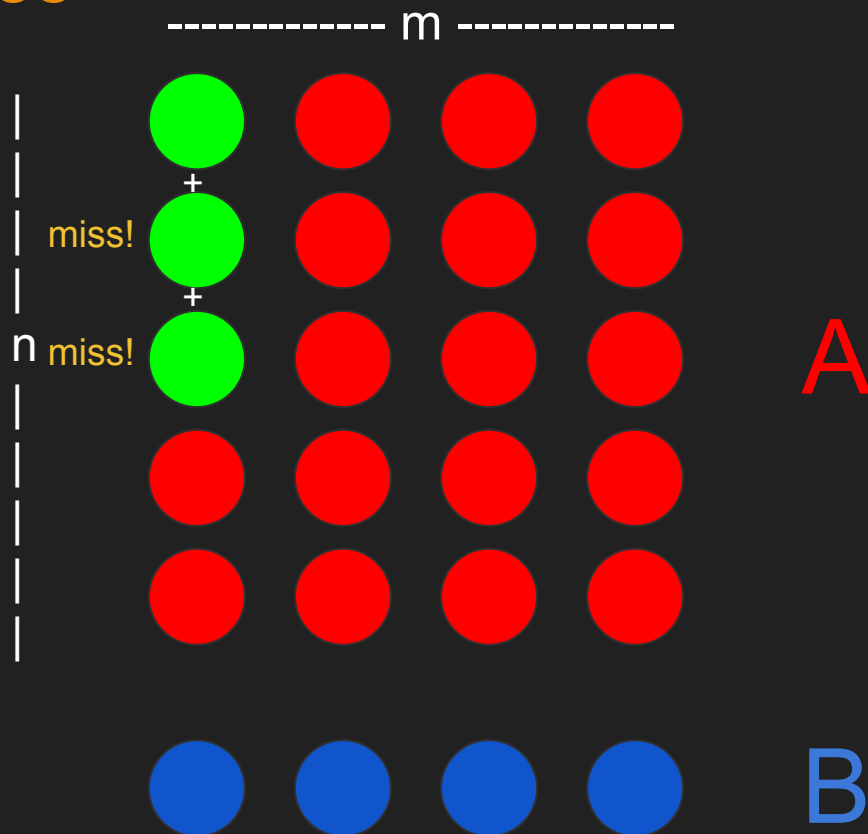
Iteration Space



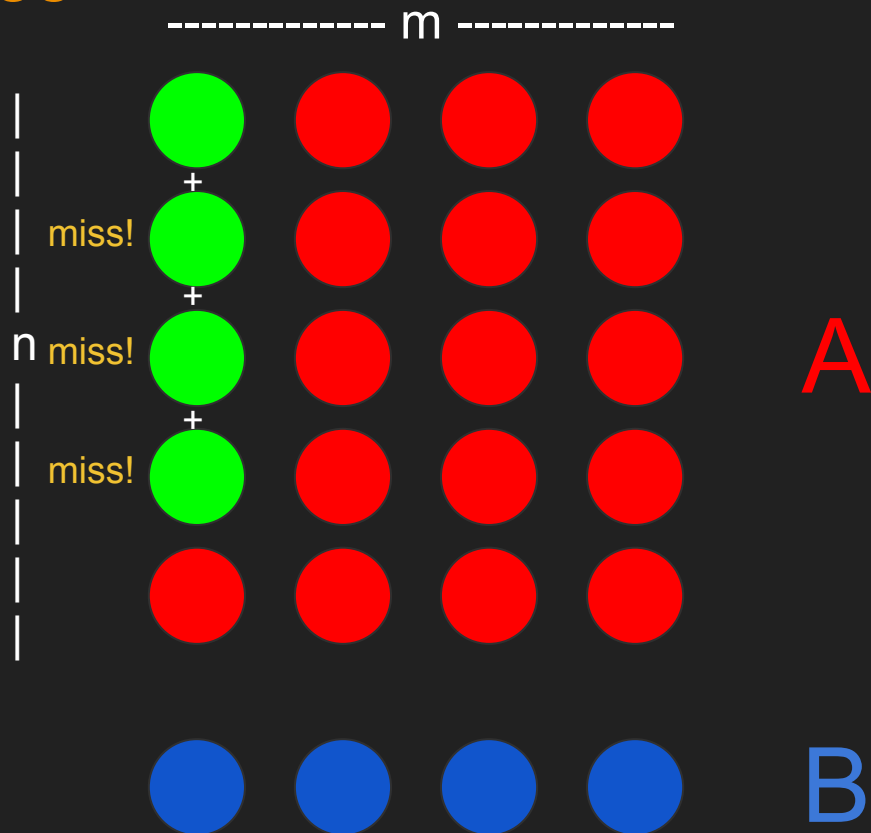
Iteration Space



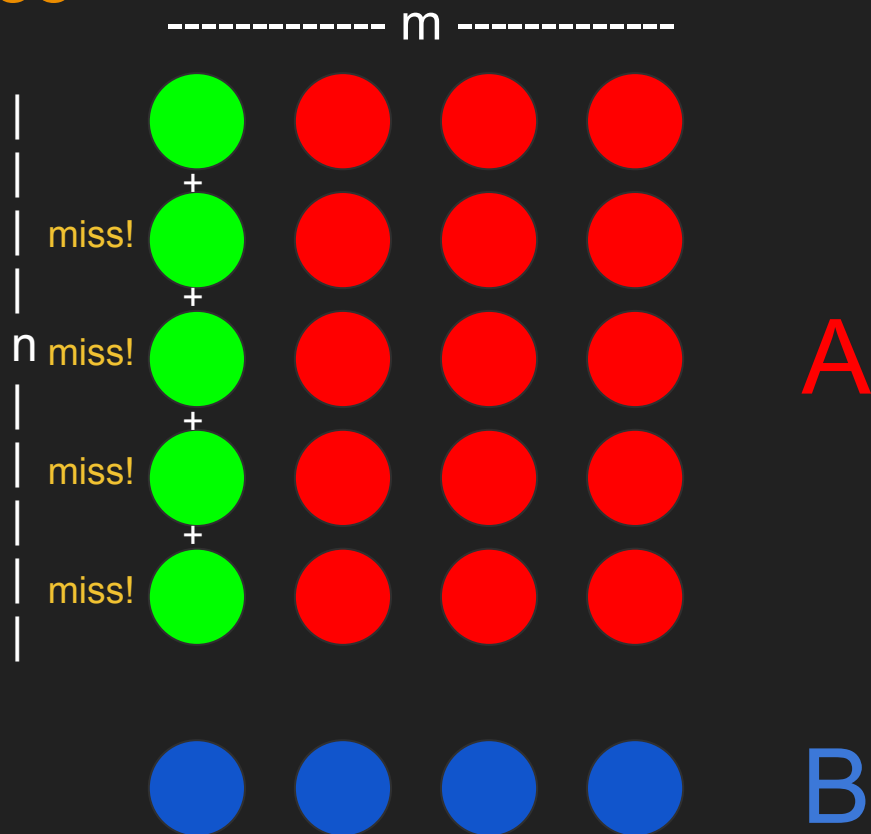
Iteration Space



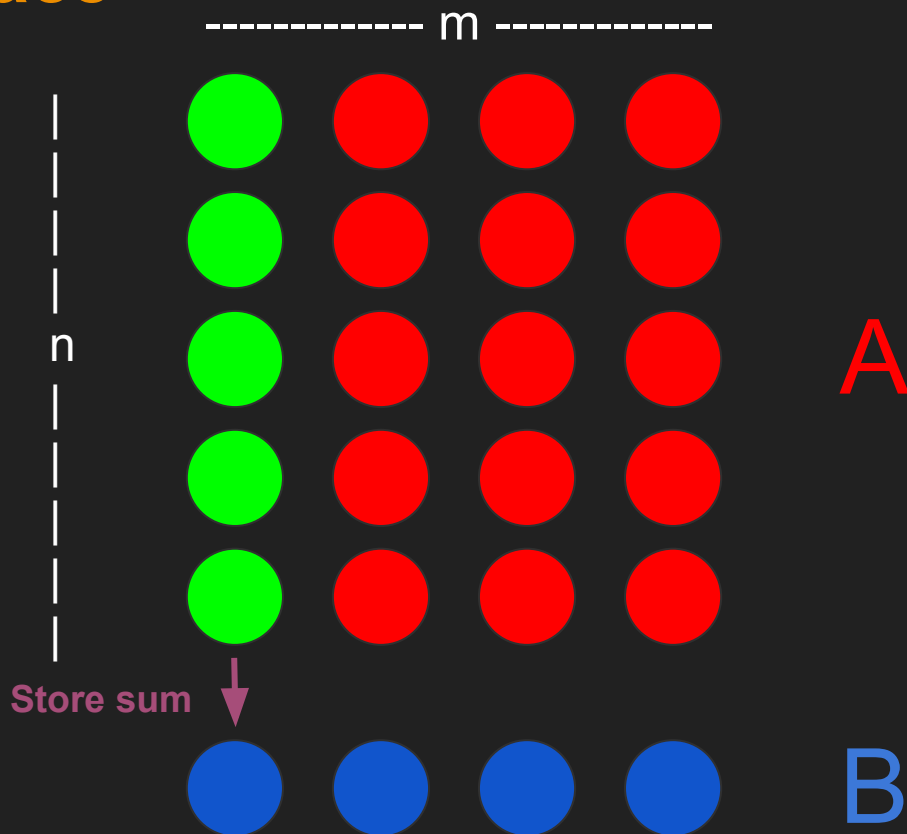
Iteration Space



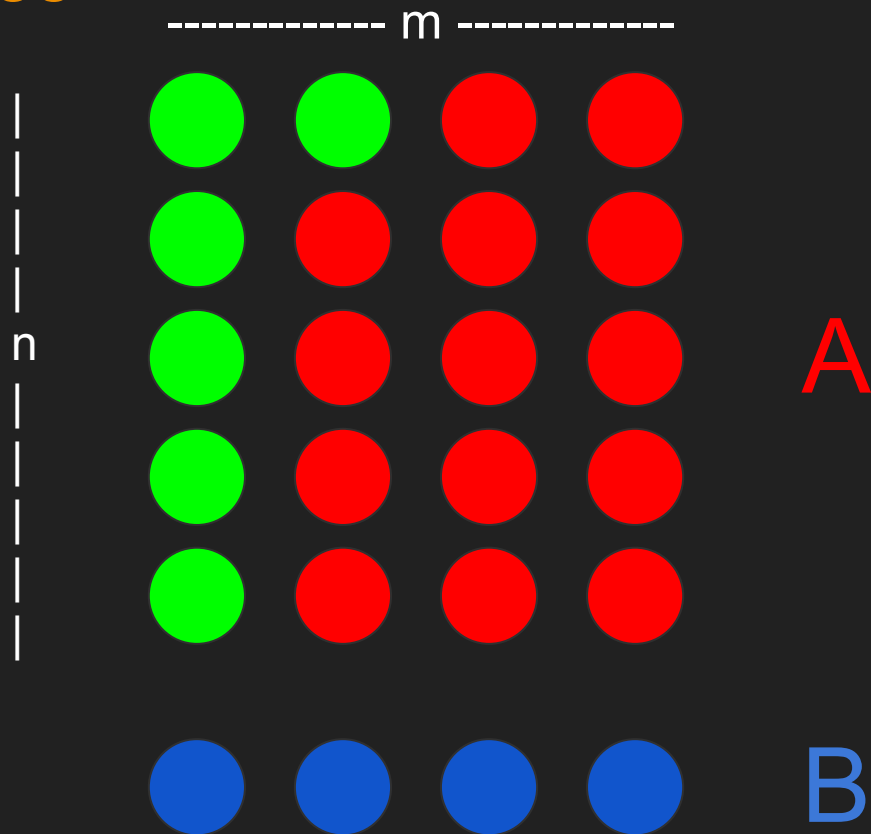
Iteration Space



Iteration Space



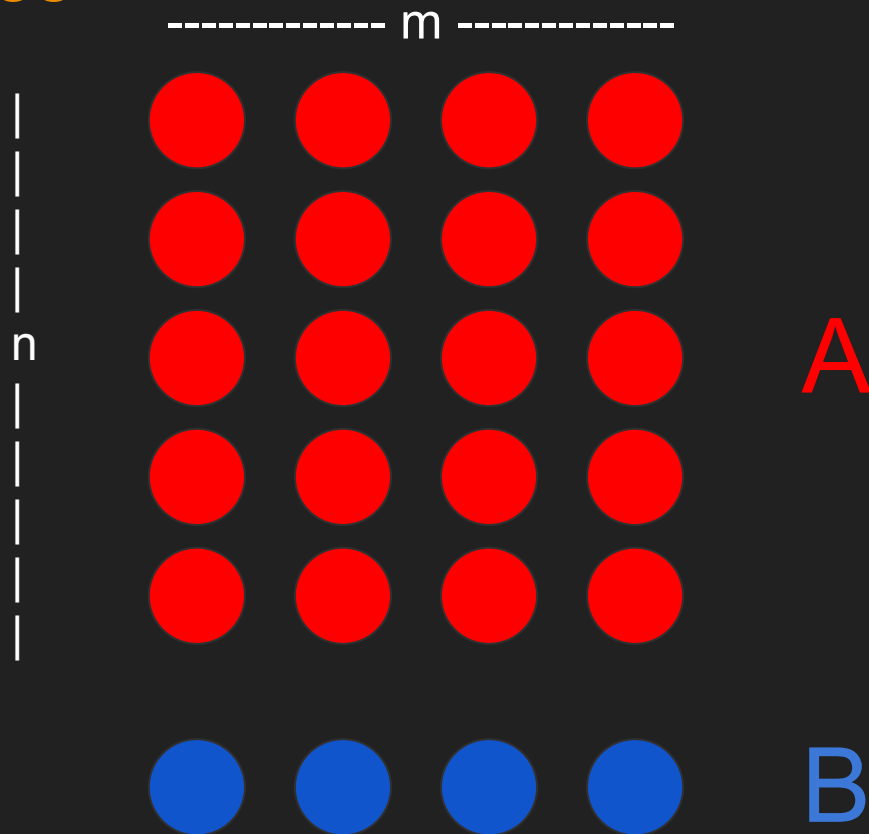
Iteration Space



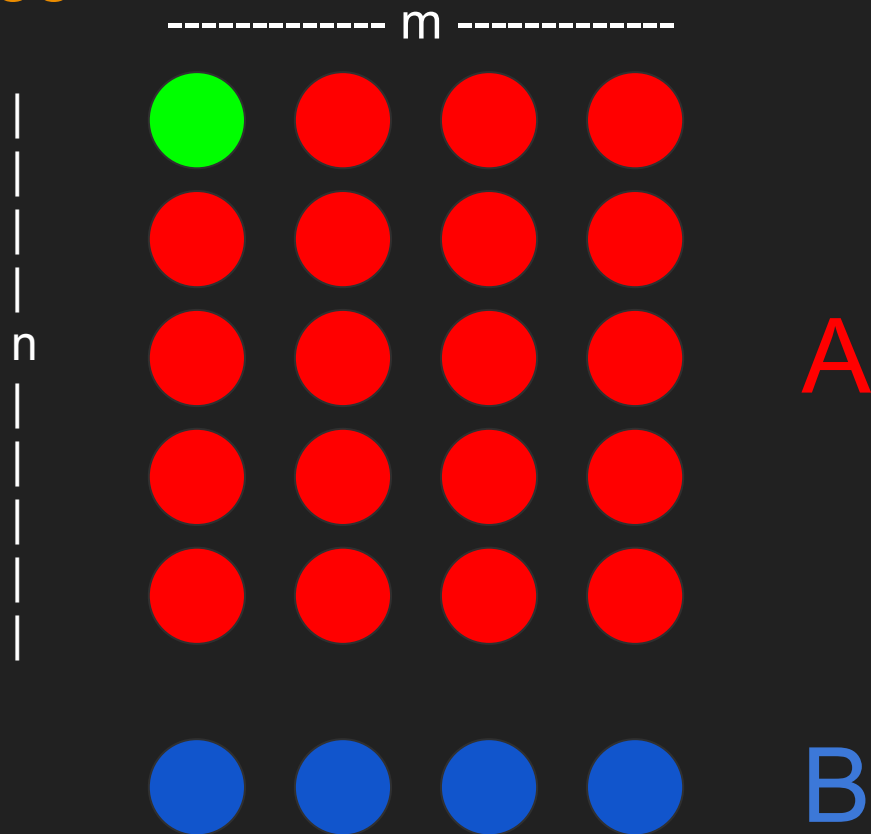
Loop Interchange?

```
... zero B;  
for (int j = 0; j < m; ++j) {  
    for (int i = 0; i < n; ++i) {  
        int v = A[j*m + i];  
        B[i] += v;  
    }  
}
```

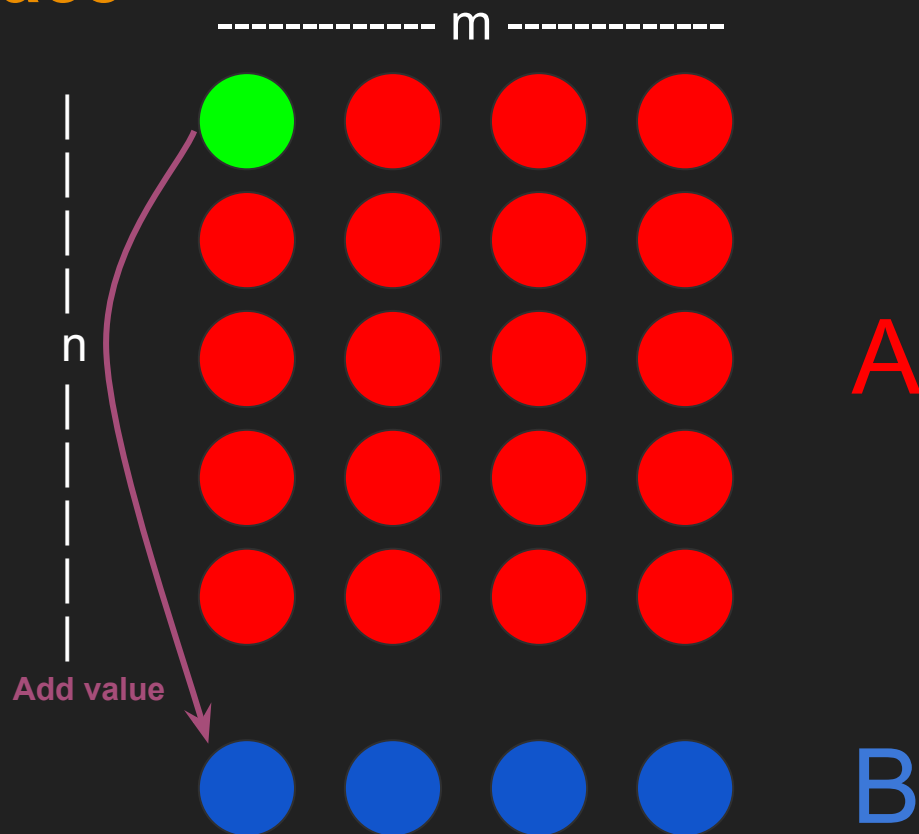
Iteration Space



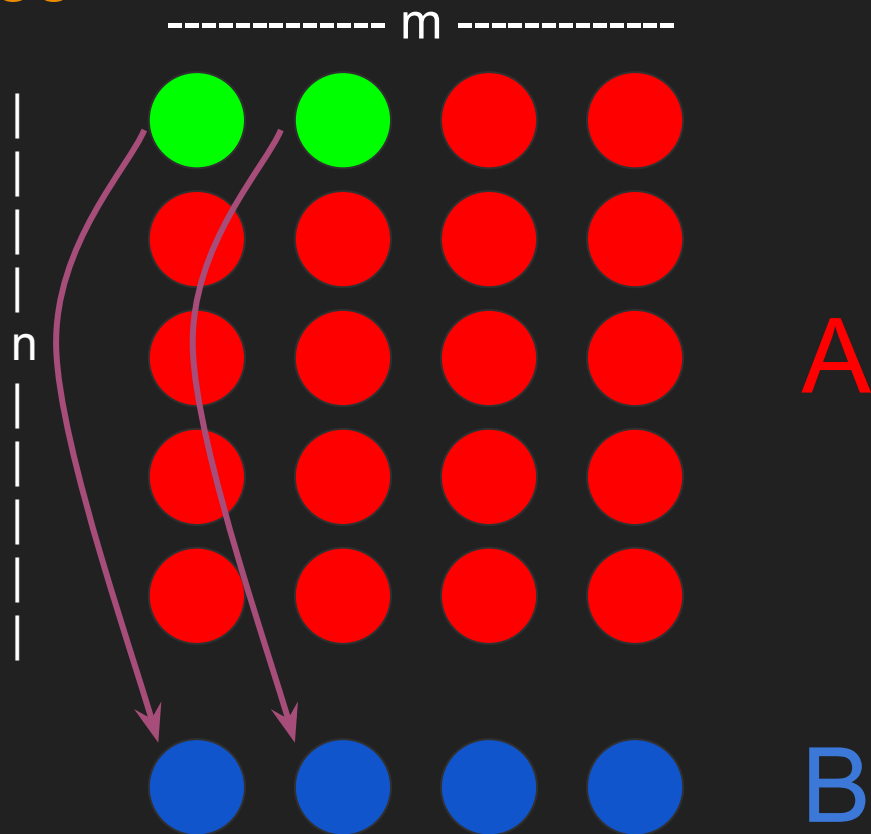
Iteration Space



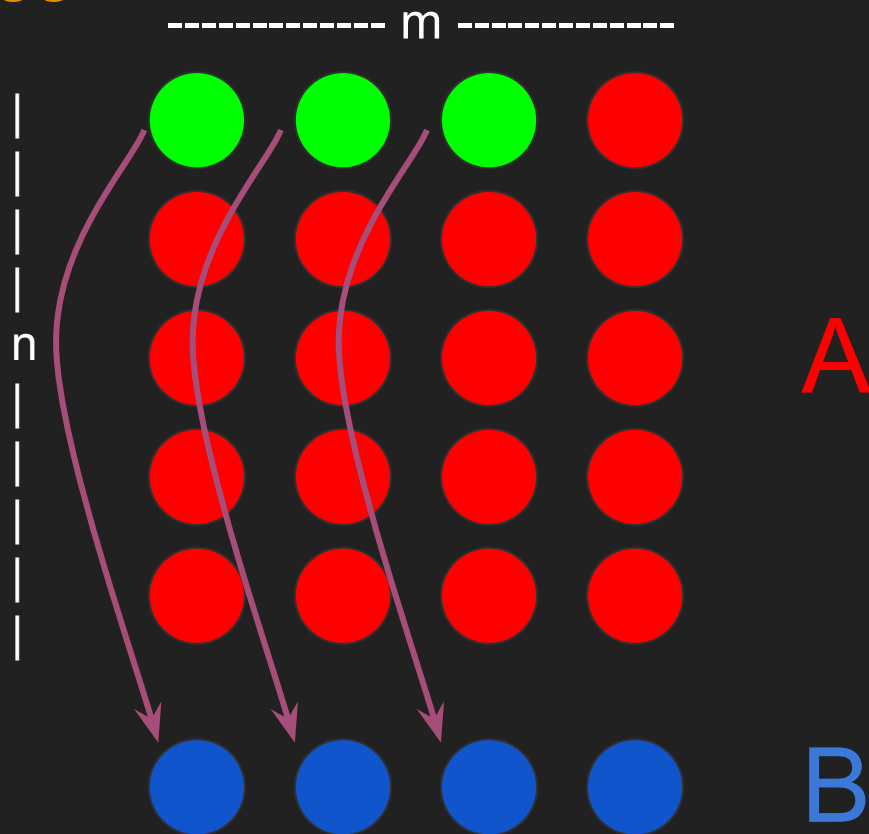
Iteration Space



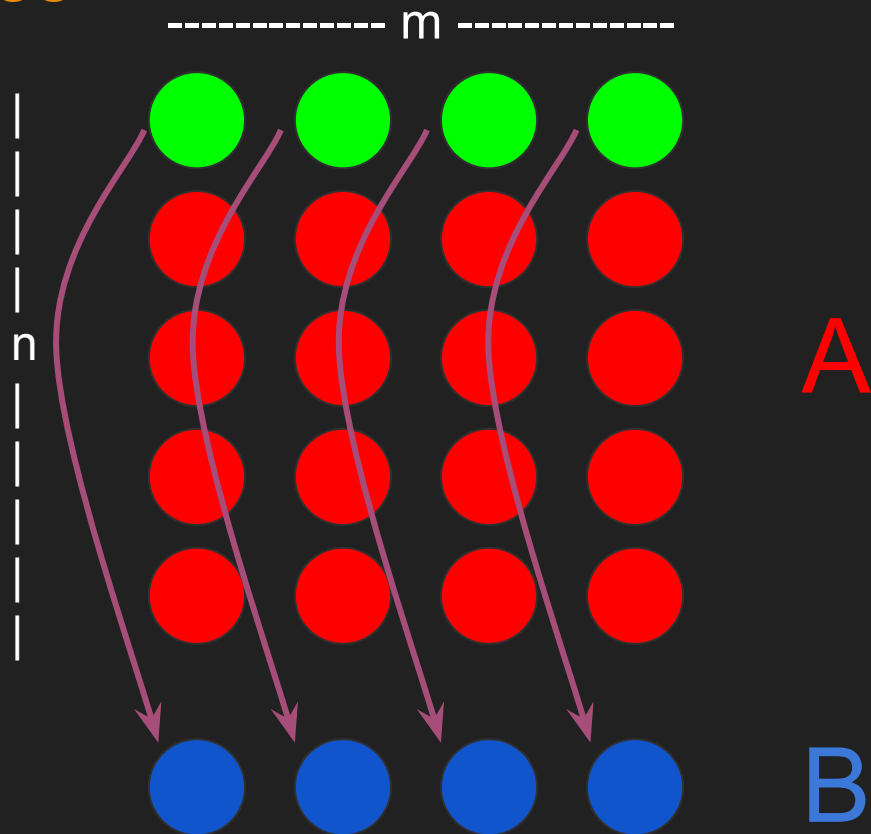
Iteration Space



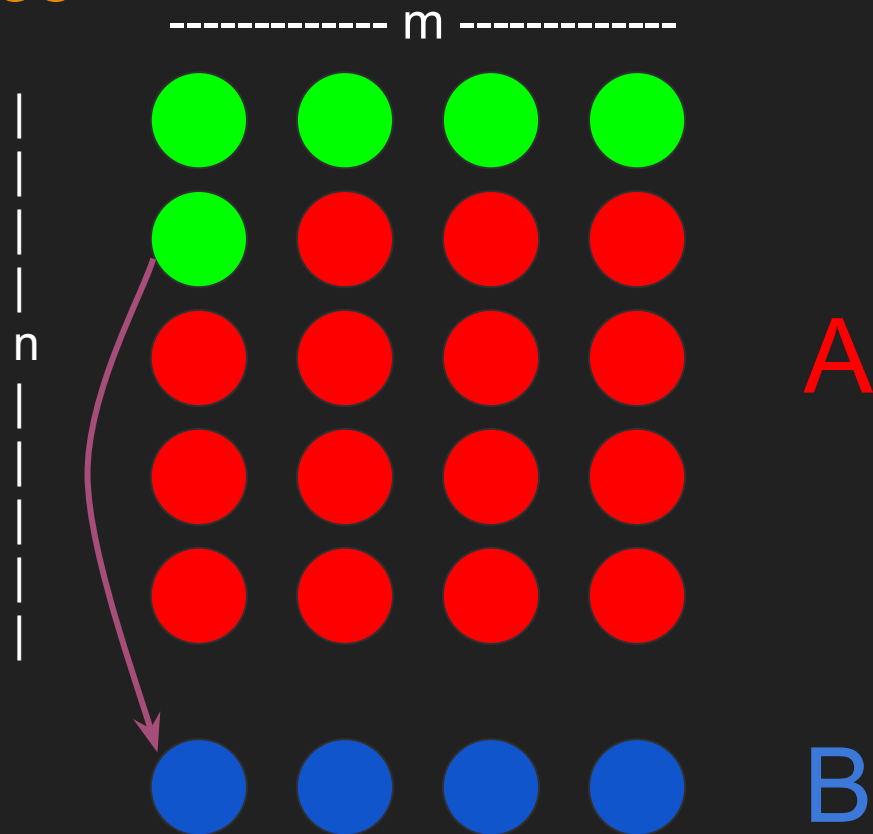
Iteration Space



Iteration Space



Iteration Space



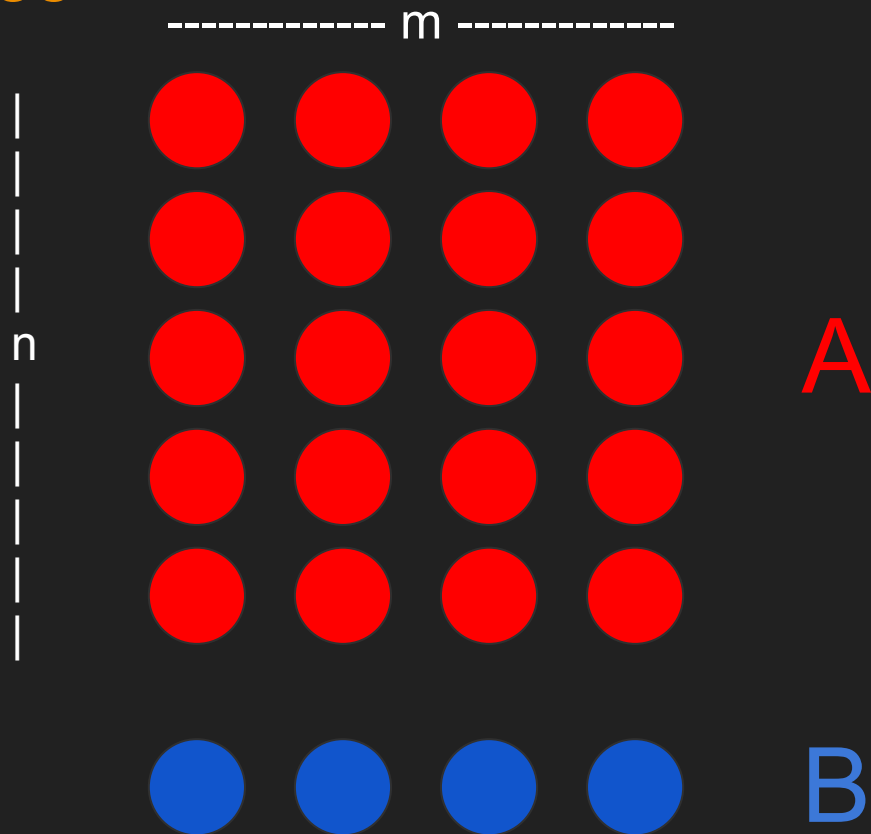
Outer-Loop Vectorization

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        int v = A[j*m + i];  
        a += v;  
    }  
    B[i] = a;  
}
```

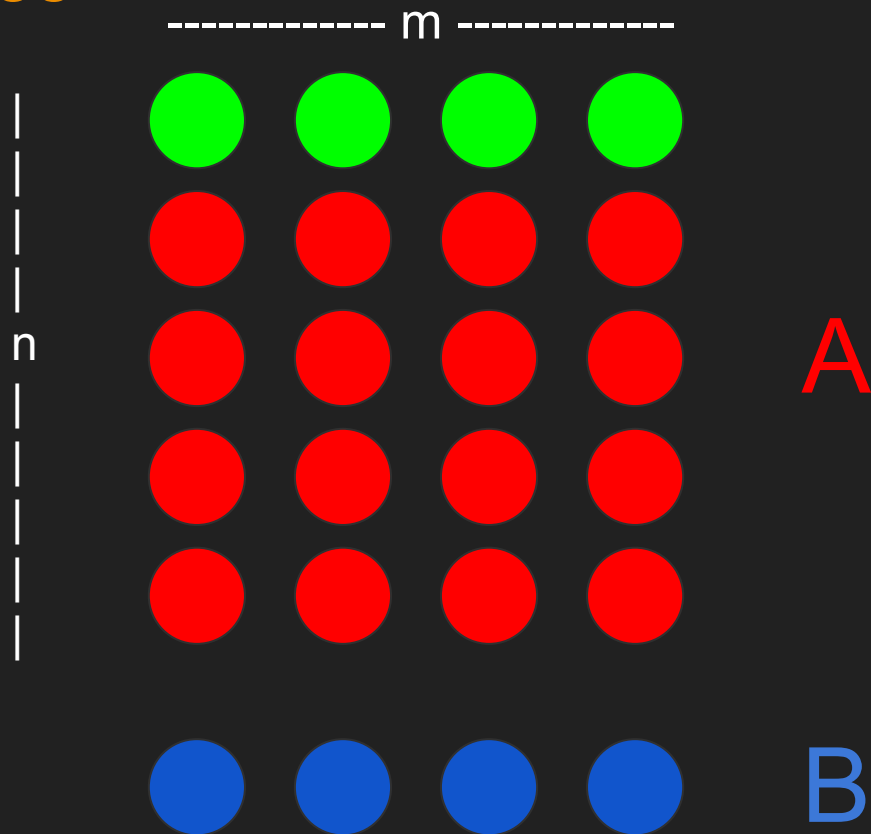
Outer-Loop Vectorization

```
for (int i = 0; i < end; i += 4) {  
    __m128i a = [0,0,0,0];  
    for (int j = 0; j < m; ++j) {  
        __m128i v = loadu_si128(A[j*m + i]);  
        a = add4(a, v);  
    }  
    storeu_si128(&B[i], a);  
}  
... residual
```

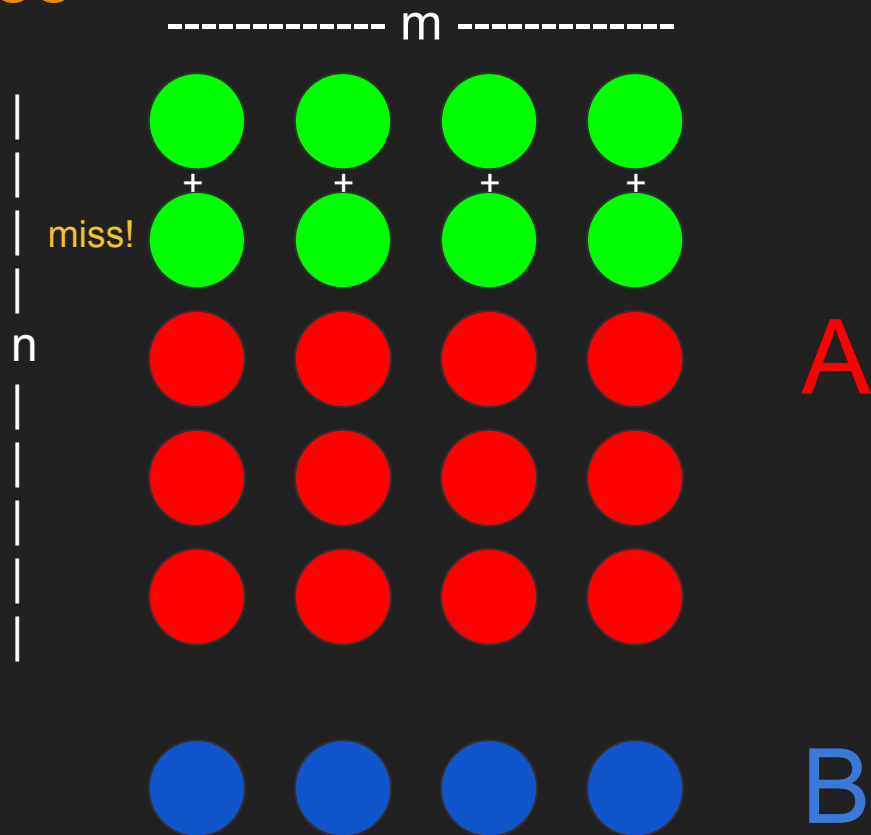
Iteration Space



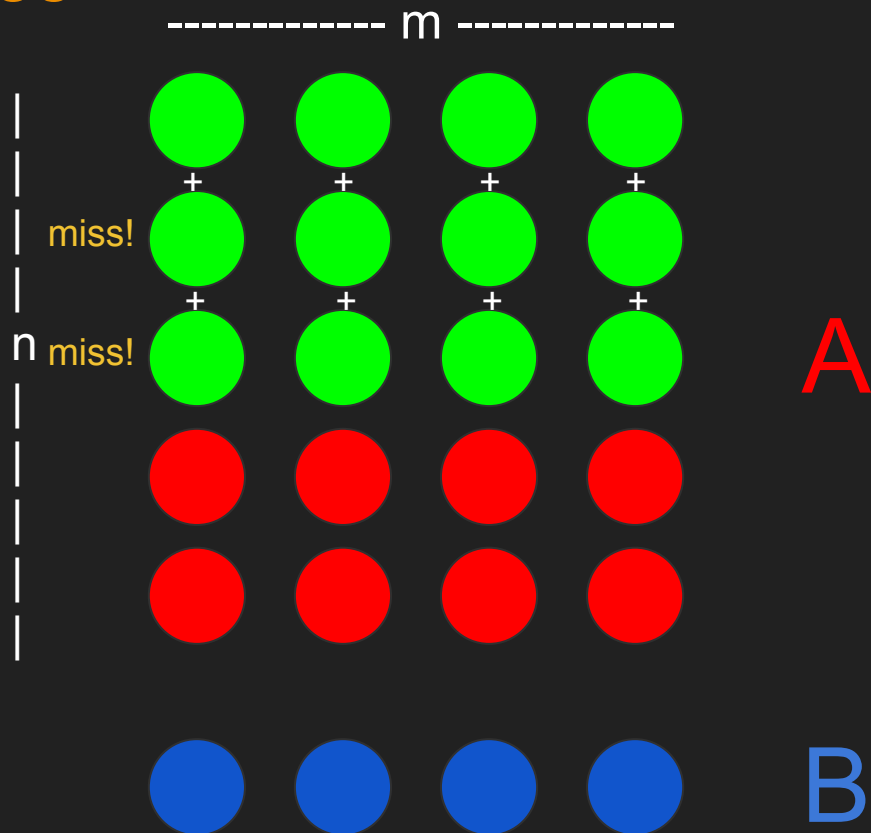
Iteration Space



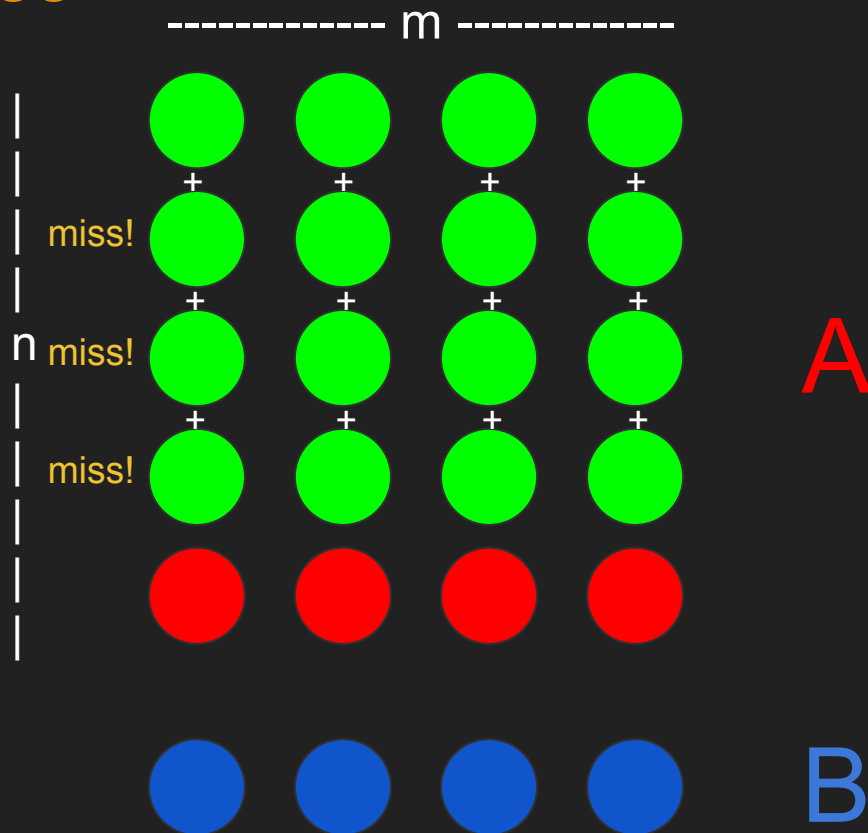
Iteration Space



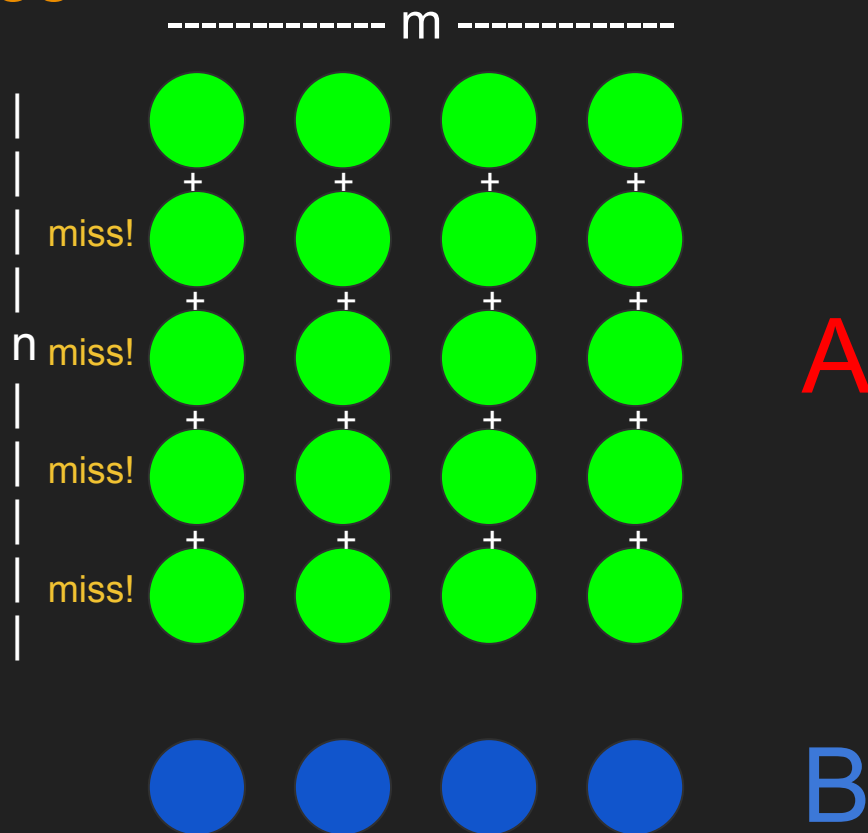
Iteration Space



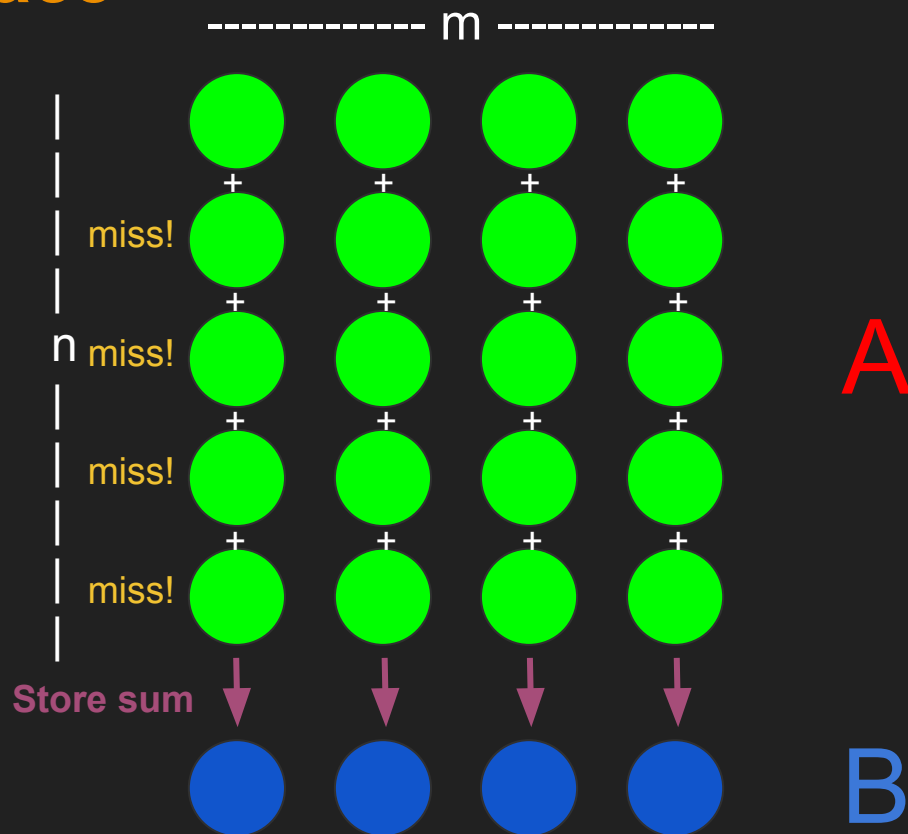
Iteration Space



Iteration Space



Iteration Space



Conditional Code

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        bool p = C[j];  
        if (p) {  
            int v = A[j*m + i];  
            a += v;  
        }  
    }  
    B[i] = a;  
}
```

Uniformity

A uniform value is characterized in relation to a loop and is one that does not vary *because of this loop*.

Divergence

Divergent = !Uniform

Uniformity in Innermost Loops

```
for (int i = 0; i < N; ++i) {  
    int v = 1 + 2;  
    a[i] = v;  
}
```

Uniformity in Innermost Loops

```
for (int i = 0; i < N; ++i) {  
    int v = 1 + 2;  
    a[i] = v;  
}
```

v is uniform and also loop-invariant

Hoist Out

```
int v = 1 + 2;  
for (int i = 0; i < N; ++i) {  
    a[i] = v;  
}
```


Conditional Code

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        bool p = C[j];  
        if (p) {  
            int v = A[j*m + i];  
            a += v;  
        }  
    }  
    B[i] = a;  
}
```

Conditional Code

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        bool p = C[j];  
        if (p) {  
            int v = A[j*m + i];  
            a += v;  
        }  
    }  
    B[i] = a;  
}
```

What is p
in the
i-loop?

Conditional Code

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        bool p = C[j];  
        if (p) {  
            int v = A[j*m + i];  
            a += v;  
        }  
    }  
    B[i] = a;  
}
```

What is p
in the
i-loop?

Uniform?



Conditional Code

```
for (int i = 0; i < n; ++i) {  
    int a = 0;  
    for (int j = 0; j < m; ++j) {  
        bool p = C[j];  
        if (p) {  
            int v = A[j*m + i];  
            a += v;  
        }  
    }  
    B[i] = a;  
}
```

What is p
in the
i-loop?

Uniform? 

Loop-Invariant? 

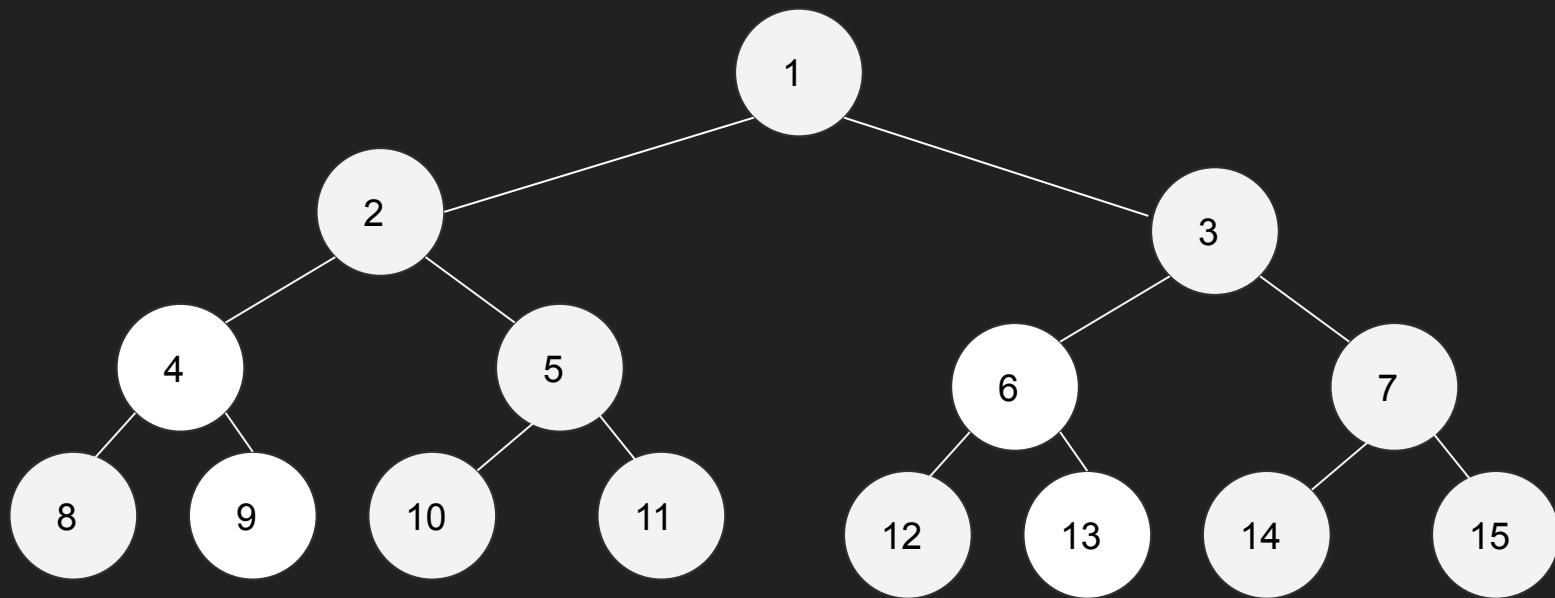
Uniformity Enables Outer-Loop Vectorization

```
for (int i = 0; i < end; i += 4) { // vectorized
    __m128i a = [0,0,0,0];
    for (int j = 0; j < m; ++j) {
        bool p = C[j];
        if (p) {
            __m128i v = loadu_si128(A[j*m + i]);
            a = add4(a, v);
        }
    }
    storeu_si128(&B[i], a);
}
... residual
```

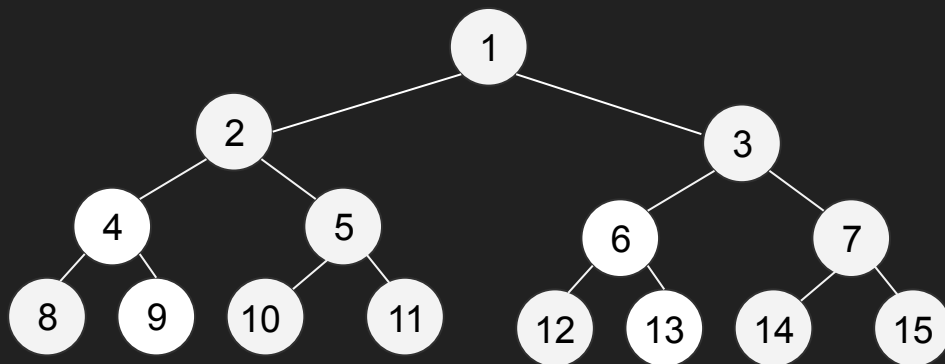
Recursive Tree Traversal Vectorization

Milind Kulkarni et al.

Tree



Tree



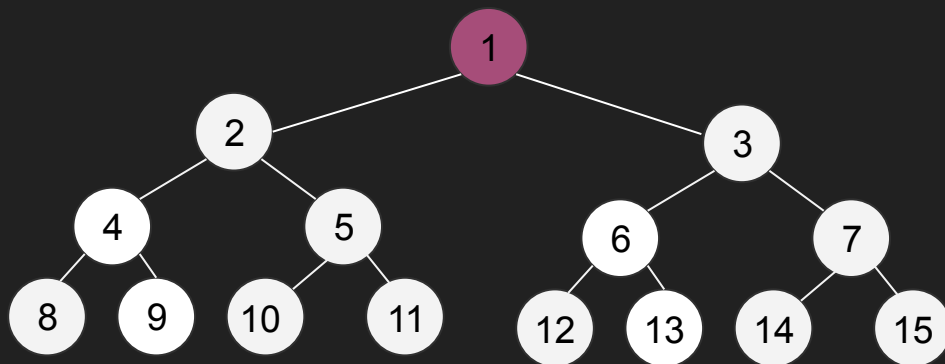
Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D

Tree



Nodes

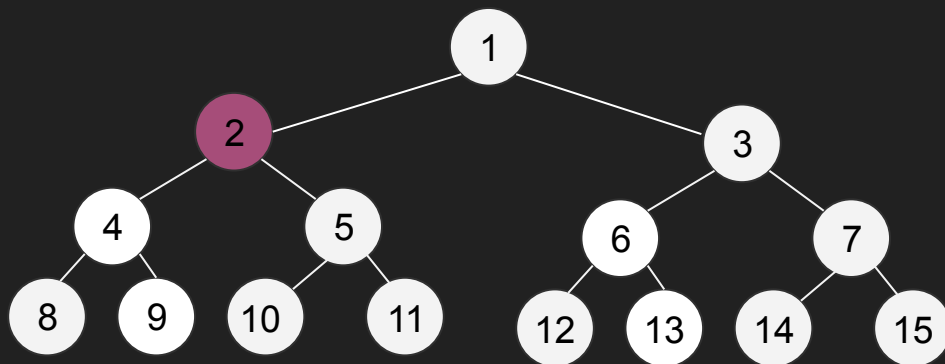
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

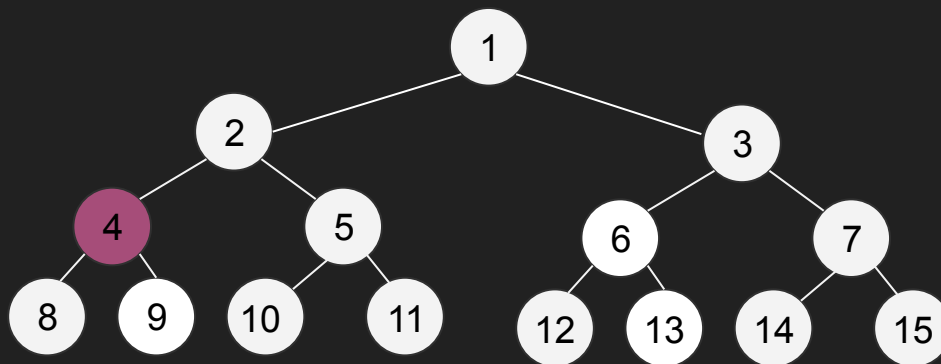
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

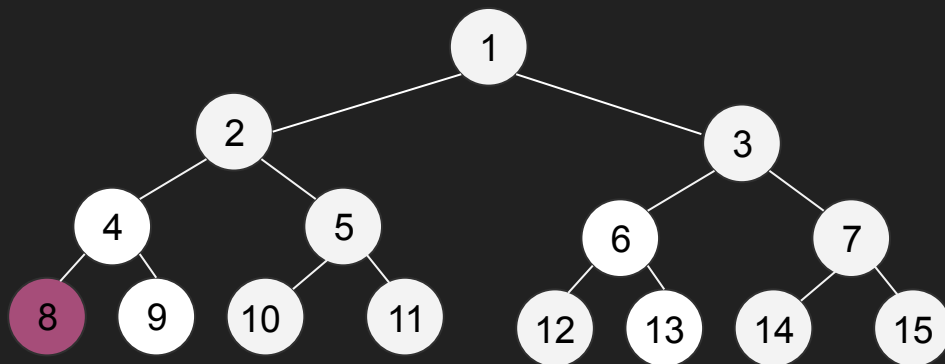
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

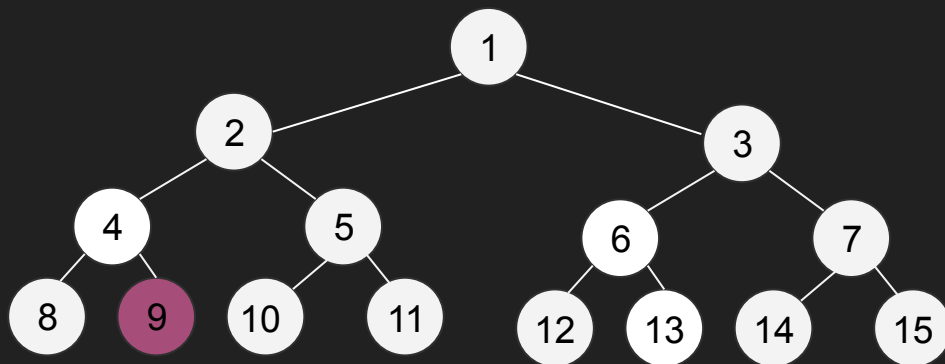
A

B

C

D

Tree



Nodes

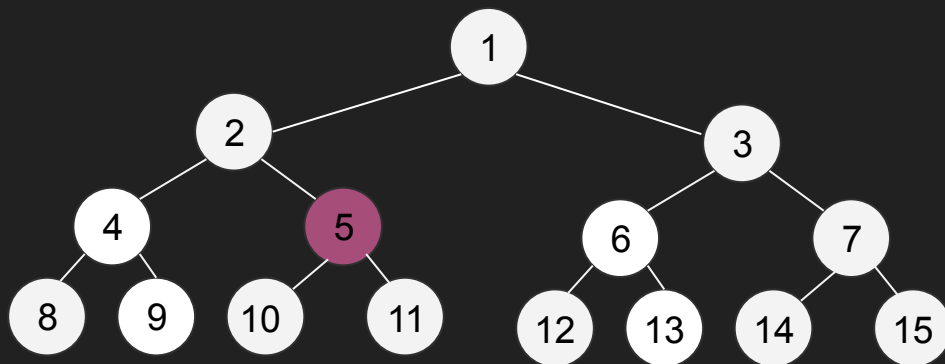
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

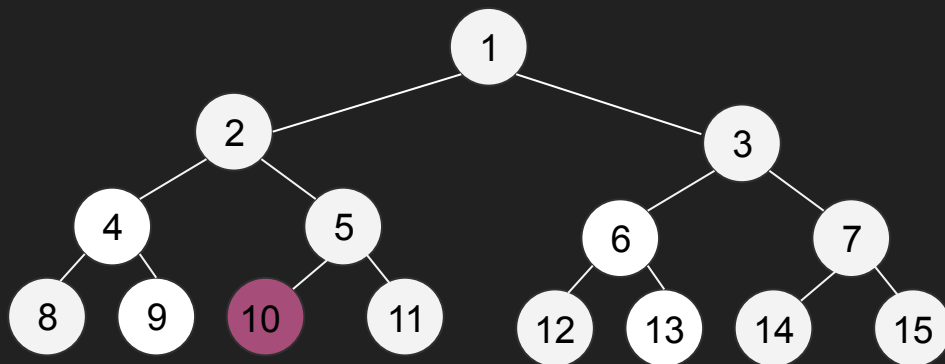
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

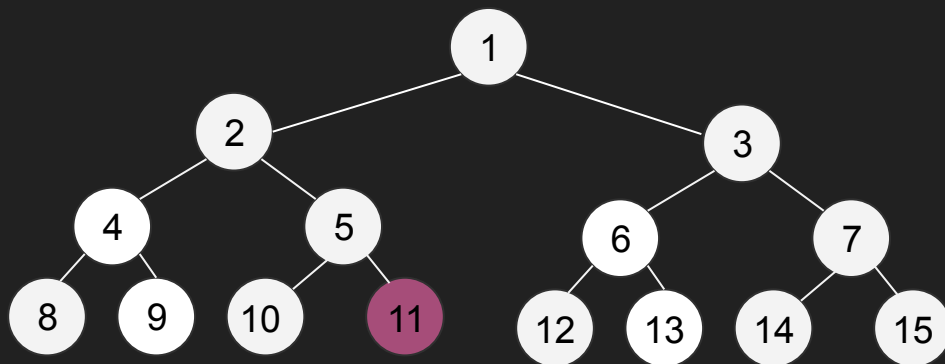
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

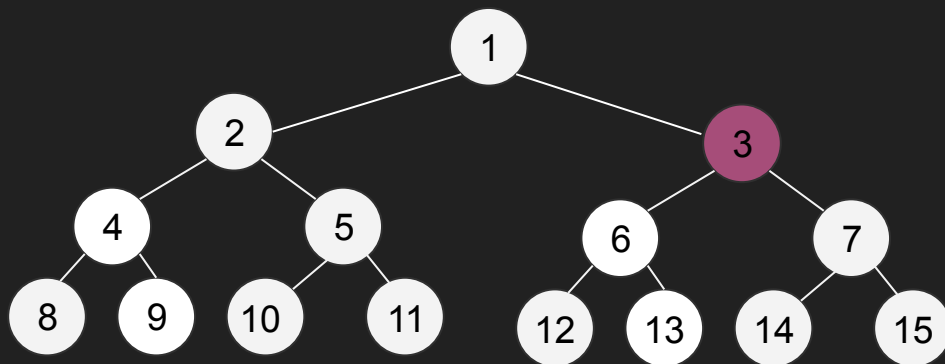
A

B

C

D

Tree



Nodes

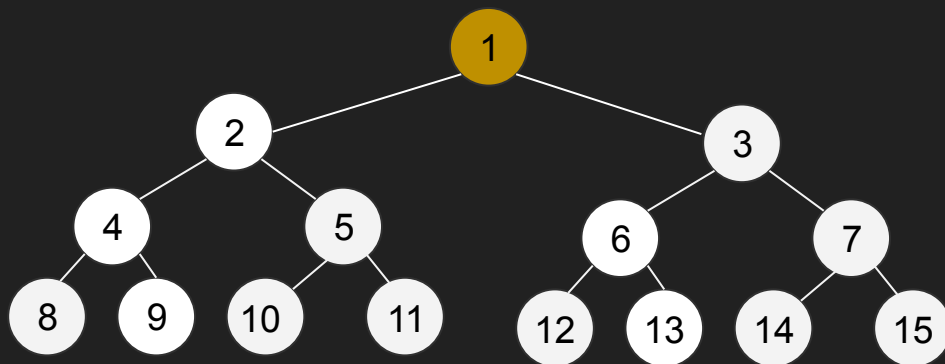
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

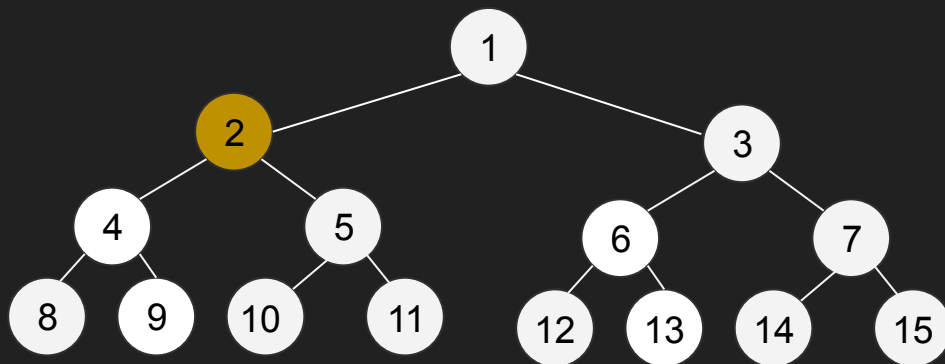
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

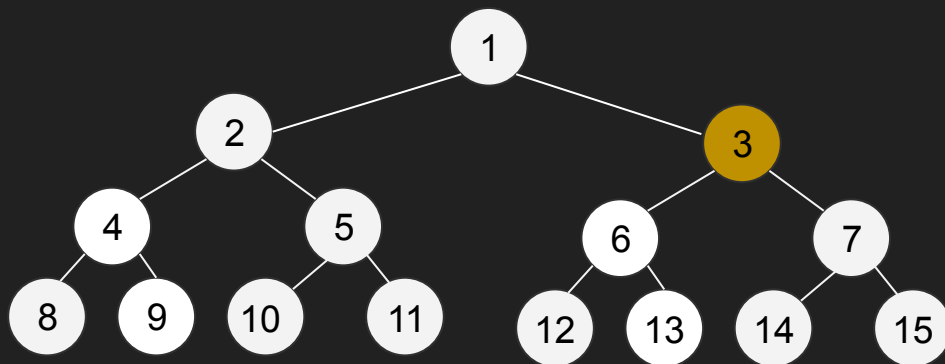
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
B
C
D



Tree



Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

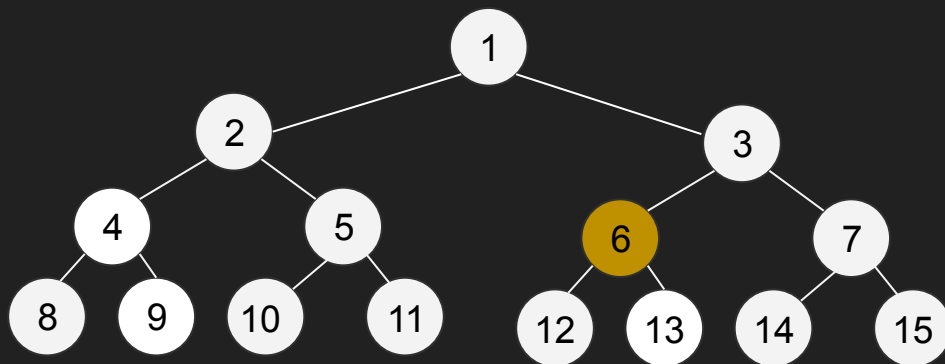
A

B

C

D

Tree



Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

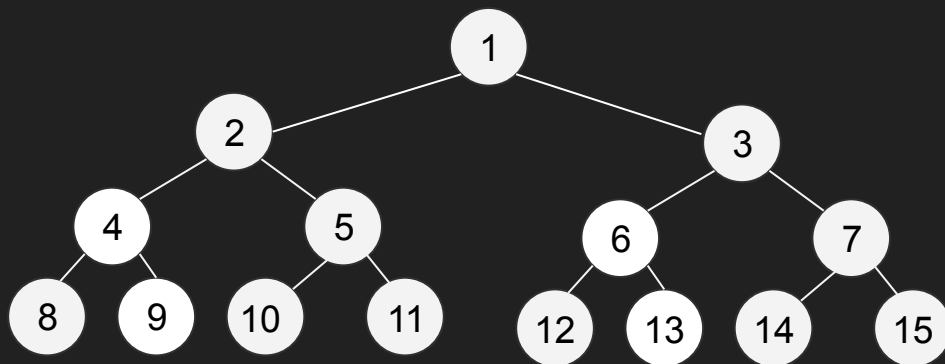
A

B

C

D

Tree



Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

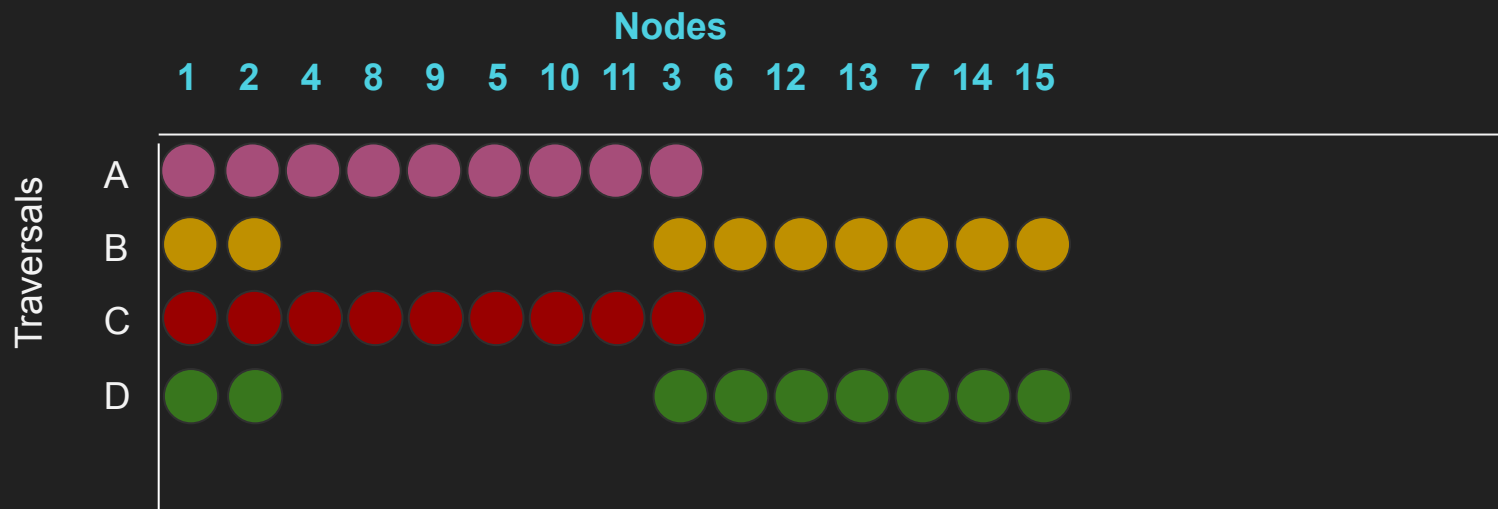
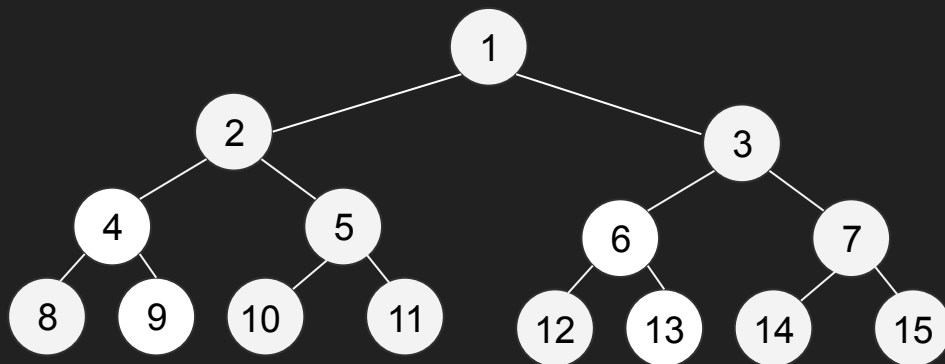
A

B

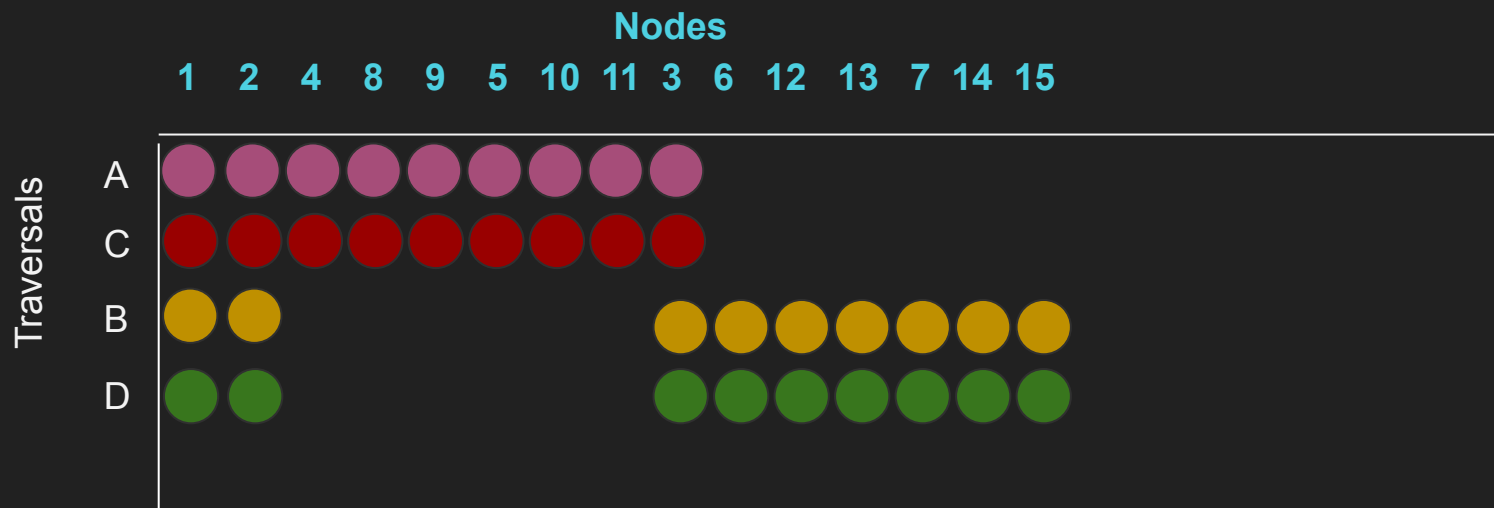
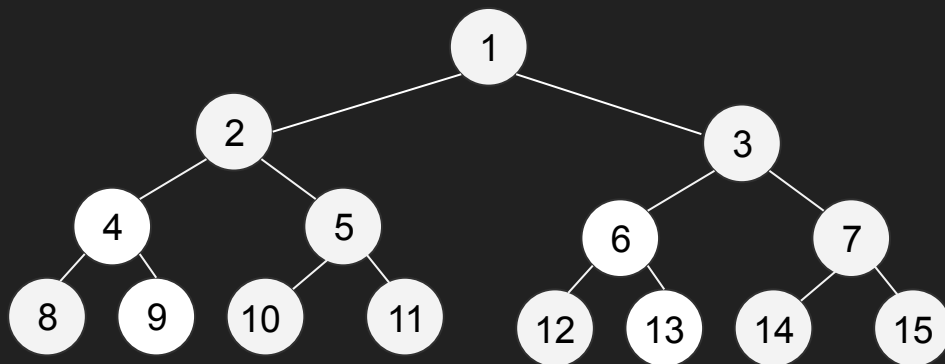
C

D

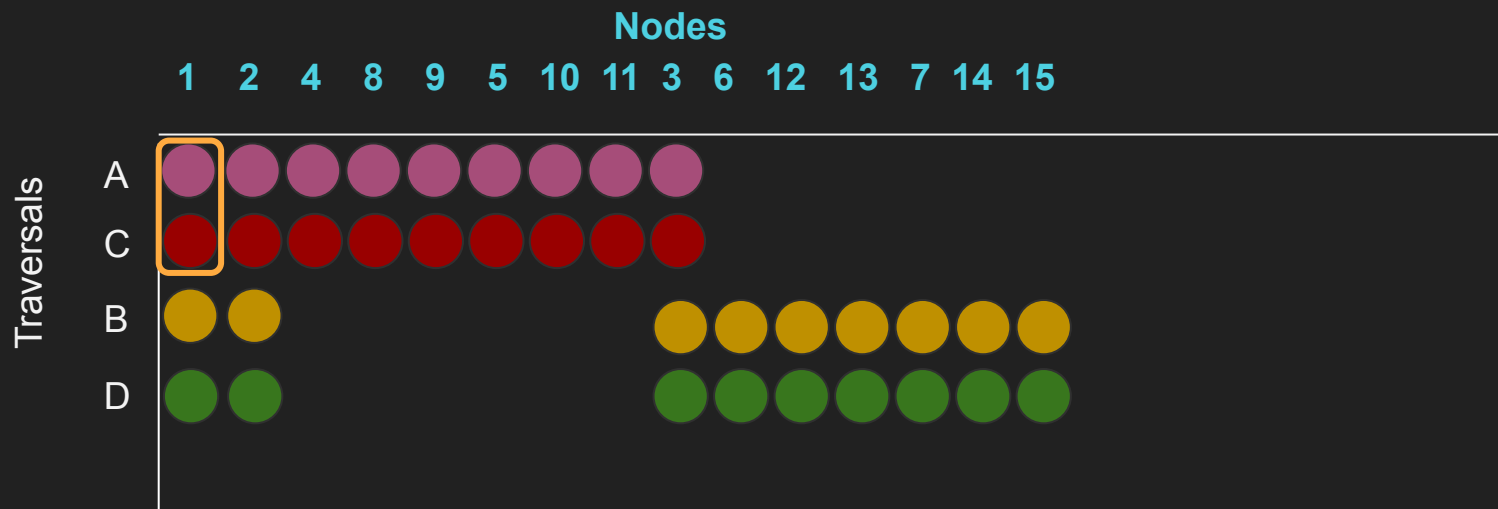
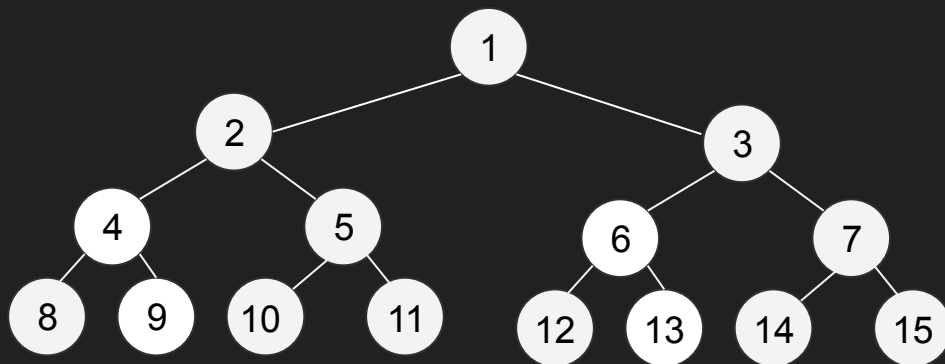
Tree



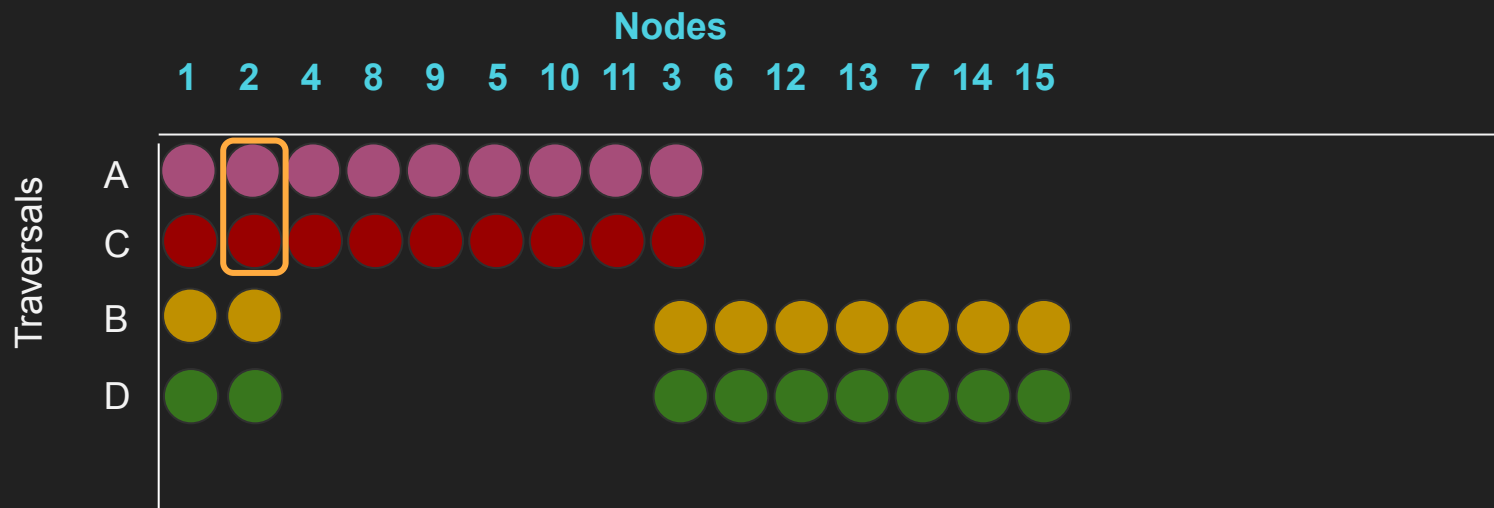
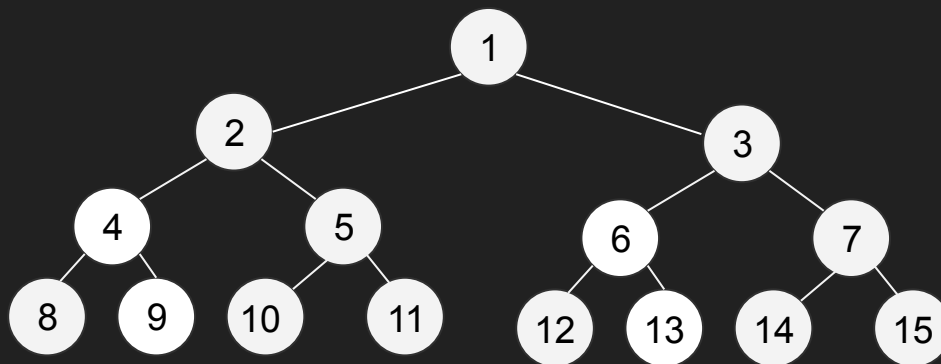
Tree



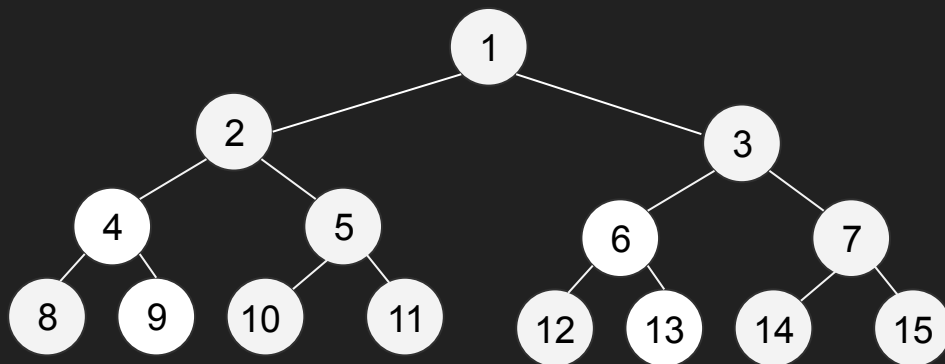
Tree



Tree



Tree

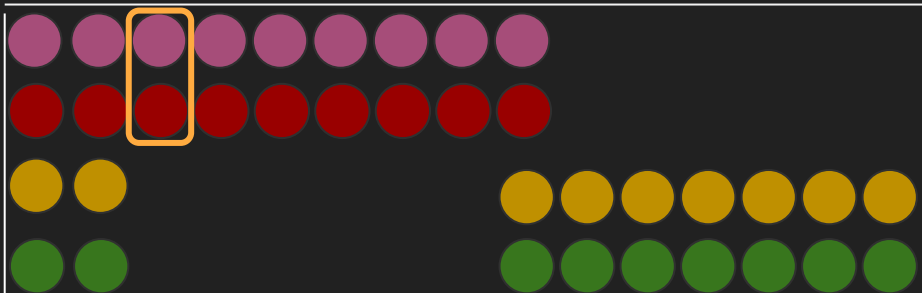


Nodes

1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Traversals

A
C
B
D



Thank You!